Loop Parallelization using Dynamic Commutativity Analysis

Loop Parallelization using Dynamic Commutativity Analysis

Christos Vasiladiotis
University of Edinburgh
United Kingdom
c.vasiladiotis@ed.ac.uk

Roberto Castañeda Lozano
University of Edinburgh
United Kingdom
rcastae@inf.ed.ac.uk

Murray Cole
University of Edinburgh
United Kingdom
mic@inf.ed.ac.uk

Björn Franke
University of Edinburgh
United Kingdom
bfranke@inf.ed.ac.uk

***Abstract*—Automatic parallelization has largely failed to keep its promise of extracting parallelism from sequential legacy code to maximize performance on multi-core systems outside the numerical domain. In this paper, we develop a novel *dynamic commutativity analysis (DCA)* for identifying parallelizable loops. Using commutativity instead of dependence tests, DCA avoids many of the overly strict data dependence constraints limiting existing parallelizing compilers. DCA extends the scope of automatic parallelization to *uniformly* include both regular array-based and irregular pointer-based codes. We have prototyped our novel parallelism detection analysis and evaluated it extensively against five state-of-the-art dependence-based techniques in two experimental settings. First, when applied to the NAS benchmarks which contain almost 1400 loops, DCA is able to identify as many parallel loops (over 1200) as the profile-guided dependence techniques and almost twice as many as all the static techniques *combined*. We then apply DCA to complex pointer-based loops, where it can successfully detect parallelism, while existing techniques fail to identify *any*. When combined with existing parallel code generation techniques, this results in an average speedup of $3.6\times$ (and up to $55\times$) across the NAS benchmarks on a 72-core host, and up to $36.9\times$ for the pointer-based loops, demonstrating the effectiveness of DCA in identifying profitable parallelism across a wide range of loops.**

***Index Terms*—commutativity analysis, parallelization.**

I. Introduction

Compilers for automatic parallelization have failed to deliver on their promise to seamlessly transition sequential legacy software into the multicore era [1], [2]. Despite intensive research, the problem of discovering and exploiting parallelism hidden in sequential code is far from solved [3], except for limited success in a few narrow domains (e.g., regular array-based computations) [4].

Researchers have moved to the use of runtime information to supplement static analyses [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23]. Using a subset of all potential inputs, these analyses are not guaranteed to be correct for every possible workload. In practice, however, it has been shown that they can be applied without sacrificing program safety [8], [9], [24].

On the other hand, recent work has shown that dependence analysis, even when informed with perfect profiling information, is inherently unable to identify any further latent parallelism [25].

A promising approach to overcome the limitations of *both* static and dynamic dependence analyses is the use of *commutativity analysis* [26], [27], [28]. In a nutshell, commutativity analysis detects code regions whose order of execution can be exchanged without affecting the program outcome.

Our key insight is that some dependences detected by static and dynamic dependence analyses are not, in fact, fatal to parallelization, and therefore cause some loops to be needlessly discarded. Instead, commutativity analysis focuses on the more crucial issue of whether any such dependences have a detectable effect on the eventual result.

In this paper, we develop a novel Dynamic Commutativity Analysis (DCA) that combines both static and dynamic information for the discovery of profitable parallelism in sequential legacy code. Unlike previous commutativity approaches, DCA focuses on loops as the program regions that typically capture most of the program execution time.

DCA handles, in *uniform* manner, both loops that are in the scope of traditional dependence approaches (such as affine loops) and loops that are beyond the capabilities of traditional methods, such as Pointer-Linked Data Structure (PLDS) traversals, which are usually dealt with *ad hoc*. This is achieved by modelling the observable live-out [29] effects of computations as invariants while simultaneously allowing other "non-essential" computations to violate transient data dependence relations.

This paper contributes a method allowing programmers to parallelize programs that are outside the scope of traditional parallelizing compilers. We also explore aspects that affect profile-driven parallelizing techniques such as the profitability and safety of parallelization. DCA improves detection of profitably parallelizable loops which are correctly identified as such even when limited, yet representative, input is used. We envision DCA being used as part of an interactive or semi-automatic parallelism advisor, where the user has the final word over any code transformations.

A. Motivating Examples

Consider the simple loop traversal over an array in Fig. 1(a), performing a *map* operation written in C. Such patterns can readily be detected as parallel using data dependences to reason about the independence of array accesses across the iteration space of this loop. However, using a different data structure to write what is in essence the same trivial code, shown in Fig. 1(b), defeats dependence analysis.

978-1-7281-8613-9/21/$31.00 © 2021 IEEE

```
1 for (i = 0; i < N; ++i) {
2   array[i]++;
3
4 }
```

(a) Array-based loop.

```
1 while (ptr) {
2   ptr->val++;
3   ptr = ptr->next;
4 }
```

(b) PLDS-based loop.

Figure 1. Simple loops that perform the same *map* operation. The right-hand side version defeats dependence analysis.

```
1  Graph *g;
2  int *dist;
3  WorkList *frontier, *next_frontier;
4  ...
5  push(frontier, g->adjacent[source]);
6  /* while there is still work in the frontier */
7  while (frontier->size) {
8    /* top-down step */
9    while (frontier->size) {
10     /* remove node from worklist to process */
11     current = pop(frontier);
12     /* go over its adjacent nodes */
13     Node *n = current->next;
14     while (n) {
15       if (dist[n->vert] > dist[current->vert]) {
16         dist[n->vert] = dist[current->vert] + 1;
17         /* this node's distance was updated,
18          * so recheck its neighbors */
19         push(next_frontier, n);
20       }
21       n = n->next;
22     } /* end of adjacent node loop */
23   }   /* end of top-down step */
24   swap(frontier, next_frontier);
25 } /* end of worklist loop */
```

Figure 2. A BFS implementation from Lonestar [30] employing complex and irregular PLDS-based loop traversals.

Dependence analysis tries to establish the *read* and *write* operations that occur at a program's memory locations and associate these locations with various relations (e.g., *read-after-write* dependence). In Fig. 1(b), the `ptr` pointer is being read in order to update the `ptr->val` data element of each node in the linked list. More crucially, `ptr` itself is being updated (i.e., read and written) to point to the next node for processing by the next loop iteration. This creates a cross-iteration *read-after-write* dependence on `ptr` which dependence analysis finds present even when profiling information is used. Hence, in this case dependence analysis is inherently incapable of determining the independence of iterations and cannot further propose this loop as a valid candidate for parallelization.

While there have been various attempts to deal with such codes (e.g., pattern matching), they have been proven inflexible and very limited against real programs. Consider Fig. 2 that shows a Breadth-First Search (BFS) graph traversal from the Lonestar benchmark suite [30]. BFS employs a worklist to iterate over all the nodes of a graph, traverses the adjacent nodes for each of them and conditionally updates their distance from the selected source node. Its result is the array `dist` containing the distance of each node from source.

The issues with parallelization of the innermost loop in Fig. 2 (lines 14–22) are similar to those discussed above. In addition, dependences between loop iterations induced by the dynamically updated `frontier` and `next_frontier` worklists via the `push` and `pop` operations prevent conventional parallelization of the top-down step.

This motivates the development of a new method that extends the automatic detection of parallelizable code in real-world programs, addressing some of the aforementioned problems. To this end, we propose Dynamic Commutativity Analysis (DCA); an analysis which utilizes liveness-based commutativity, instead of dependences, to detect potentially parallel loops. By using the live-out variables of a loop (i.e., those consumed later in the program), DCA focuses on the parts of the computation that have an impact on its outcome, irrespective of its traversal idiom. Moreover, using liveness, DCA observes if the outcome remains unaffected when permuting the iterations of a loop, thus strongly suggesting that this loop's iterations can be executed in any order, or in fact in parallel with appropriate synchronization. In other words, we say that DCA detects the loop as *commutative*.

For example in Fig. 2, the significant result of the outermost loop is the variable `dist`, which is indeed found as live-out after line 25. In this case, DCA determines that this update loop (lines 9–23) is also *commutative* and can subsequently be executed in parallel. This is because processing of the `current` nodes produces the same `dist` values, regardless of the order it occurs.

Overall, DCA detects more potential parallelism in complex and diverse loops in a *uniform* manner, overcoming obstacles which thwart dependence analysis.

B. Contributions

This paper makes the following contributions:

- We introduce Dynamic Commutativity Analysis (DCA), a technique for testing the commutativity and hence potential parallelizability of arbitrarily complex loops, the program regions with the highest parallelization potential, including traversals of Pointer-Linked Data Structures.
- We show that DCA is substantially more effective in detecting parallelizable loops from a wide range of benchmarks, outperforming a *combination* of state-of-the-art dependence-based approaches.
- We demonstrate that our approach is able to discover loops with significant parallelization profitability and high precision.

C. Overview

The rest of this paper is structured as follows: Section II reviews existing commutativity concepts, Section III describes our notion of liveness-based loop commutativity, Section IV describes DCA, Section V evaluates our approach, Section VI discusses related work, and Section VII concludes.

II. Background

A number of different notions of commutativity for parallelization can be found in the literature [26], [27], [28]. We briefly discuss them here to set the scene for our novel analysis in Section IV.

A. Separability-Based Commutativity

A pioneering notion of commutativity has been first introduced in [26]. It views computation as composed of separable operations on objects, where each operation has a receiver object and several parameters that are passed by value to it. If all the operations required to perform a given computation commute without affecting the final result, then the compiler can generate parallel code. This approach quickly reaches its limits for real-world applications, since it requires programming in a very restrictive object-oriented style. Later work has focused on verifying commutativity conditions for linked data structures [31], limited to operations that produce semantically equivalent states in different execution orders.

B. Output-Based Commutativity

[27] proposes an alternative notion of commutativity analysis for individual functions, which considers the *output* of a function at its point of use. A candidate function is symbolically executed in two different call orders to create an abstract representation of the result for each order. This symbolic result is then used as input to all functions that could potentially read it, and, in turn, these "reader" functions are symbolically executed. If the outputs of these reader functions are identical, then the initial function is commutative. Whilst this notion of commutativity has strength in handling e.g., unordered container data structures in the same spirit as [31], it is limited to the repeated, possibly commutative invocations of a single function.

C. Liveness-Based Commutativity

Commutativity based on liveness [29] has been introduced in [28], and used as part of a formal characterization of parallel algorithmic skeletons. The *key idea* here is to restrict commutativity requirements to only those variables, which are live-out at a region of interest. This allows for the separation of transient variables (and associated computations), whose values are not used anywhere later in the code, to be relaxed. While developing the concept, the paper did not provide an analysis for the *liveness-based commutativity* property. In this paper we seek to provide a practical dynamic analysis for this notion of *liveness-based commutativity* when applied to loops and demonstrate its use in parallelizing real-world sequential applications.

D. Applicability of Parallelization

Identifying a code region (i.e., function, loop, etc.) as commutative does not immediately guarantee its correct parallel execution.

The separability-based commutativity model in [26] already fulfills a lot of the requirements for parallelization as a natural consequence of its strict computation model and also employs a series of conservative checks to guarantee the safe parallelization of a method. For output-based commutativity [27], the step to parallelism is not clearly described and the evaluation takes place in a simulation environment with infinite-issue capability.

At this stage in our approach, we combine DCA's profile-guided nature with reemployment of data dependences to extract parallelism and guide synchronization, similarly to [8]. Wherever safe parallelization cannot be conclusively determined, we lean on the user for final approval (Section IV-D).

III. Liveness-Based Loop Commutativity

The notion of commutativity has a natural interpretation for loops: a loop is commutative if rearranging its iterations preserves the outcome of the original program. This section introduces our notion of *liveness-based loop commutativity*. We consider loops, permutations of their iterations, and the values of variables which they touch. Ideally, for a loop to be declared commutative, we would like to check that, given some values for its live-in variables [29], permuting its iterations has no impact on the values of its live-out variables. This is clearly infeasible due to a combinatorial explosion of possible permutations and inputs, and a similar problem would impede the use of symbolic techniques, despite recent progress [32]. We take a more pragmatic approach, described fully in section Section IV, checking for a selected set of iteration permutations, and with representative inputs. Interestingly, our results (Section V-D) show that in practice there is a close correspondence between this pragmatic commutativity checking scheme and the more comprehensive version described above.

IV. Dynamic Commutativity Analysis

Our analysis consists of a static and a dynamic stage. Fig. 3 depicts the components of our approach. We have also implemented a parallelization stage to evaluate the effectiveness of our scheme. To easily follow DCA's operation, we provide examples in C of the intermediate code generated by each step in Fig. 4, for the simple loops of Fig. 1.

A. Static Stage

Our analysis goes over the source of an input program and selects loops for further processing. For every loop nest, each loop is considered separately by a series of compiler passes operating on intermediate representation code.

1) Iterator/Payload Separation: One of the crucial aspects of this stage is our ability to detect which parts of the loop form the *iterator* code and which the actual computation, i.e., the *payload*, using a *generalized iterator recognition* analysis [33]. Intuitively, this analysis identifies the set of variables that are updated on each iteration *and* determine if execution continues in the loop body or exits out of it.

This allows the static stage of DCA to tackle a range of non-affine loop iterators, while avoiding the use of a limited *ad hoc* scheme. We have already discussed the challenges of dealing with a wider range of iterator styles in Section I-A. Fig. 4(a) highlights which lines would be identified and *separated* as iterator from the code of Fig. 1.

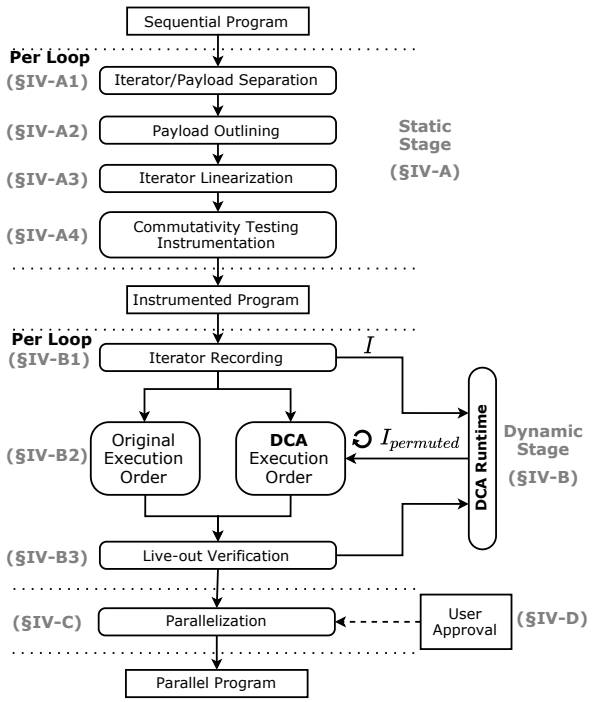

Figure 3. Overview of Dynamic Commutativity Analysis.

2) Payload Outlining: Next, we outline the *payload* part of the loop in a separate function. This allows us easier handling of the code section in subsequent stages and in particular during execution when coupled with our instrumentation. We identify the live-in, live-out and live-through variables of the outlined region and provide them to the outlined function as arguments. By construction the iterator values are consumed as live-in variables. Fig. 4(b) shows the resulting code after outlining.

3) Iterator Linearization: We proceed by instrumenting the identified iterator code in order to extract its values during profiling. This process *linearizes* our subsequent accesses to values of the iterator and is similar in spirit to the linearization described in [34]. Our variant is able to tackle a wider range of loop idioms and data structure traversals as it is powered by a *generalized iterator recognition*.

Fig. 4(c) shows the code from Fig. 4(b) after inserting calls to the DCA runtime library. While this is depicted in a separate standalone loop (lines 1–4) for simplicity, in practice it can simply be accomplished by proper placement of these calls in the loop header when operating with low-level intermediate representation code.

4) Commutativity Testing Instrumentation: This stage concludes by placing additional calls to our runtime, enabling commutativity testing of the loop (Fig. 4(d)). These calls serve two basic purposes: (i) permute the order of the loop iterations, and (ii) verify the commutativity property of the tested loop. Their operation during execution is discussed in detail in the next section. The final output is an instrumented program along with auxiliary reports on the loops that were transformed.

```
1 for (i = 0; i < N; ++i) {
2   array[i]++; // payload
3
4 }
```

```
while (ptr) {
  ptr->val++; // payload
  ptr = ptr->next;
}
```

(a) Iterator/Payload Separation.

```
1 for (i = 0; i < N; ++i) {
2   payload1(i, array);
3
4 }
5
6 void
7 payload1(int it, int *v1) {
8   v1[it]++;
9 }
```

```
while (ptr) {
  payload2(ptr);
  ptr = ptr->next;
}

void
payload2(struct node *it) {
  it->val++;
}
```

(b) Payload Outlining.

```
1 for (i = 0; i < N; ++i) {
2   rt_iterator_linearize(i);
3
4 }
5
6 for (i = 0; i < N; ++i) {
7   payload1(i, array);
8
9 }
10 ...
```

```
while (ptr) {
  rt_iterator_linearize(ptr);
  ptr = ptr->next;
}

while (ptr) {
  payload2(ptr);
  ptr = ptr->next;
}
...
```

(c) Iterator Linearization.

```
1 for (i = 0; i < N; ++i) {
2   rt_iterator_linearize(i);
3
4 }
5
6 rt_iterator_permute();
7
8 while(rt_iterator_next()) {
9   payload1(
10      rt_iterator_get(),
11      array);
12 }
13 ...
14 rt_verify();
15 ...
```

```
while (ptr) {
  rt_iterator_linearize(ptr);
  ptr = ptr->next;
}

rt_iterator_permute();

while(rt_iterator_next()) {
  payload2(
      rt_iterator_get()
      );
}
...
rt_verify();
...
```

(d) Commutativity Testing Instrumentation.

Figure 4. Code output for the intermediate steps (top to bottom) of DCA's static stage for an array-based (left) and a PLDS-based loop (right) respectively.

B. Dynamic Stage

During the dynamic stage of our analysis, the instrumented program produced by the static stage is executed multiple times using different configuration modes applied by our runtime library. The goal of this stage, in the spirit of Section III, is to determine commutativity for a loop by executing its iterations in different orders and comparing the resulting outcome with the one obtained by the original, programmer-intended execution order (i.e., a "golden" reference).

1) Iterator Recording: Once a DCA-transformed loop is reached during execution, the linearization step records the iterator values, using a random-access sequence container. Next, a set of permutation schedules is selected which control the exact reordering of the loop iterations. The originally prescribed order is executed by default for every loop under test since the output is required as a reference for comparing against the subsequent permuted executions.

2) DCA Execution: Exhaustively executing, using input data from the benchmark suite, all possible permutations for a set of iterator values is exponentially expensive for loops

with a large trip count. Therefore, further addressing the scalability issues discussed in Section III, we provide reduced permutation presets (e.g., reverse or a configurable number of random shuffles). This means accepting a chance of missing a commutativity violating permutation, i.e., that our dynamic analysis is generally not safe. However, as evidenced and discussed in Section V-D, this trade-off is still surprisingly powerful in practice.

3) Live-out Verification: The final step of the analysis tests the candidate loop for commutativity. After the execution of loops in the preselected iteration permutation schedules is complete, the runtime compares the produced output computation (i.e., live-outs). If the output of at least one permuted execution differs from the original output ("golden" reference) we mark the loop as non-commutative.

C. Parallelization

While the focus of this work is mainly on the detection of parallelizable loops, we have also implemented a simple parallel code generation scheme to allow evaluation. We achieve significant speedups for PLDS-based loops (Section V-B2).

Our strategy is limited at loop-level parallelism and employs the same techniques as described in [8] concerning identification of variables for privatization and reduction operations. Profiling information is used to detect privatizable variables per loop-level by following the reader and writer statements for each memory location. The exploitation of reduction operations uses the approach outlined in [35]. The produced parallel code uses the OPENMP framework which lends itself naturally and easily to this scope of parallelism, since it does not involve any other high-level code restructuring.

D. Safety

As already mentioned in Sections I to III, profile-guided parallelization cannot inherently guarantee correctness for *every* potential concrete input. However, it overcomes the overly conservative nature of static analysis, unlocking more potential parallelism. In our system, we let the user approve the cases where correct parallelization is not conclusively guaranteed.

In this work, we have also studied the rate of loop misclassification (false positives) and found that DCA correctly identifies all the reported loops as parallelizable (Section V-D). This agrees with prior research which shows that the occurrence or absence of *potentially* parallelization-inhibiting dependences are fairly stable across different program inputs [24]. As noted above, we have used inputs provided with the benchmark suites.

E. Challenges and Limitations

Candidate loops can be deeply nested, thus we explore commutative loops hierarchically in a top-down fashion, using one loop per test invocation. We mitigate this by executing several test instances at the same time.

A candidate loop can appear in different execution contexts (e.g., different call sites of containing function) during application runtime. Loop candidates can exhibit commutativity in some execution contexts, but not in others. Currently, our analysis is not context-sensitive. We leave this for future work.

TABLE I
NAS PARALLEL BENCHMARK (NPB) LOOPS REPORTED AS PARALLELIZABLE BY THE BASELINE DYNAMIC APPROACHES, DEPENDENCE PROFILING [8] AND DISCOPOP [9], AND AS COMMUTATIVE BY DCA.

Benchmark	Loops (#)	DEPENDENCE PROFILING (#)	DISCOPOP (#)	DCA (this work) (#)
BT	182	168	176	168
CG	47	33	21	33
DC	105	—	—	41
EP	9	6	8	6
FT	42	36	34	36
IS	16	12	20	12
LU	186	160	164	160
MG	81	48	66	48
SP	250	233	231	233
UA	479	—	—	466
Total	**1397**	**696**	**720**	**1203**

Execution of regions in permuted order can lead to unpredictable behavior if those loops are not commutative. We reliably detect these situations.

Generally, we assume that candidate loops do not contain I/O statements or produce any other side effects not captured by liveness (volatile memory accesses, etc.). Any such loops are excluded during the selection step of the static stage.

V. EMPIRICAL EVALUATION

We evaluate DCA's efficacy and the performance obtained by simple parallelization of the detected commutative loops against dynamic (Section V-B) and static (Section V-C) approaches. We also study aspects of its profitability and precision (Section V-D), along with its potency at the loop scope and beyond against expert parallelization (Section V-E).

A. Experimental Setup

Benchmarks. We use the NAS Parallel Benchmark (NPB) suite [49] (NPB 3.3, SNU 1.0.3) to evaluate array-based loops. The suite contains ten programs with a total of 1397 loops (listed in Tables I and III). The NPB programs implement numerical analysis kernels, in both sequential and OPENMP versions, written in C (originally derived from FORTRAN). We also use a diverse selection of programs that employ PLDS-based loops across several benchmark suites, listed in detail in Table II. Olden [42] benchmarks employing recursion were rewritten in imperative form as in [38].

Inputs. We use input workload class *B* for NPB programs, except for *MG* and *IS* which use class *C*. For the selection of programs with PLDS loops, we use the biggest workload, when available, otherwise we provide a custom workload to also enable a long-running execution for the sequential version.

Compilers. DCA was prototyped on the LLVM compiler infrastructure [50]. We also use five state-of-the-art dependence-based tools as a baseline to compare with our approach. These are:

- DEPENDENCE PROFILING [8]: a profile-driven dependence-based parallelism detection approach targeting loops.
- DISCOPOP [9]: another profile-driven dependence-based approach aiming at code regions of varying granularity.

TABLE II
PLDS-BASED LOOPS WHICH DCA DETECTS AS COMMUTATIVE AUTOMATICALLY, WHILE EXISTING PARALLELIZATION TECHNIQUES FAIL TO IDENTIFY ANY. THESE LOOPS CONTAIN PROFITABLE PARALLELISM THAT WAS EXPLOITED MANUALLY BY PREVIOUS WORK. THESE TECHNIQUES HAVE EITHER REPORTED THE SPEEDUP OVER THE SEQUENTIAL EXECUTION TIME OF THAT LOOP ONLY (LOOP) OR OVER THE WHOLE PROGRAM EXECUTION TIME (OVERALL).

Benchmark	Origin	Loop-Containing Function	Profitability			Detection Technique
			Sequential Coverage (%)	Potential Speedup (×)		Expert Manual
				Loop	Overall	
429.mcf	SPEC CPU2006 [36]	`refresh_potential`	30	2.2	—	DSWP variant 1 [37], [38]
300.twolf	SPEC CPU2000 [39]	`new_dbox_a`	30	1.5	—	DSWP variant 2 [40]
ks	PtrDist [41]	`FindMaxGpAndSwap`	99	1.5	—	DSWP variant 1
otter	FOSS	`find_lightest_geo_child`	15	2.5	—	DSWP variant 2
em3d	Olden [42]	`compute_nodes`	100	∼ 2	—	DSWP variant 1
mst	Olden	`BlueRule`	100	1.5	—	DSWP variant 1
bh	Olden	`walksub`	100	2.75	—	DSWP variant 1
perimeter	Olden	`perimeter`	100	2.25	—	DSWP variant 1
treeadd	Olden	`TreeAdd`	100	—	∼ 7	Partitioning [43]
hash	Shootout	`ht_find`	50	—	∼ 4	Partitioning
BFS	Lonestar [30]	`BFS`	99	—	21	Galois [44]
ising	*community*	`main`	95	—	∼ 6	ASC [45]
spmatmat	SPARK00 [34]	`main`	89	—	∼ 4	APOLLO [46]
water-spatial	SPLASH3 [47], [48]	`INTERF`	63	—	2	OPENMP

- IDIOMS [51]: a constraint-based analysis focusing on the detection of complex reduction and histogram operations.
- POLLY [52]: a polyhedral transformation framework.
- INTEL ICC [53]: a mature industrial compiler that uses data dependence analysis and supports auto-parallelization.

IDIOMS and POLLY are also implemented on LLVM, which allows a loop-by-loop comparison. ICC was also used to obtain loop profiling information across the benchmark programs that we assessed and the INTEL OPENMP runtime library for all parallelized execution runs. The DISCOPOP results are taken from the literature since the tool is not available.

Configuration. To meet our evaluation goals we configured the tools with the following criteria in mind: (i) during detection, maximize the identification capability for each tool regardless of profitability, (ii) while during code generation, maximize the profitable parallelization exploitation for each tool. Hence, we disable ICC's parallelization profitability heuristic (i.e., `par-threshold` option) for detecting parallelizable loops, and maximize it for the code generation phase. POLLY's profitability heuristic is disabled during detection (via the flag `-polly-process-unprofitable`) and it is only applied during code generation. All the static tools compile using optimization level `O2` with loop unrolling and vectorization disabled.

Hardware. All our execution experiments were conducted on an INTEL Xeon Gold 6154 (Skylake) CPU with 72 cores, running at 3 GHz with 540 GB RAM on Ubuntu 18.04.4 LTS (Linux kernel 4.15.0–91). Reported speedups from parallelization are an average of 5 runs. The Coefficient of Variation (CV) was 5% or less for all execution time measurements, except for *CG* generated by POLLY where the CV was 40%.

Other. We verified DCA's output on NPB using the internal suite verification routines. For the PLDS programs, we used a combination of their reference output and custom profiling.

ICC failed to compile a parallelized version of *UA*, so we use its sequential execution time in all further speedup measurements. Both DISCOPOP and DEPENDENCE PROFILING did not report results on *DC* and *UA*.

B. Performance against Dynamic Techniques

1) Detection of Array-Based Loops: Our experimental results on almost 1400 loops of the NPB suite show that DCA is effective at detecting commutative array-based loops. This is significant, since further experiments (Section V-D) provide evidence that all the detected loops are indeed comprehensively commutative and hence parallelizable. Given this close correspondence, the rest of this section also refers to commutative loops as *potentially parallelizable*.

For each benchmark, Table I reports the total number of loops and the number loops that each baseline dynamic approach reports as parallelizable, and compares these loops with the loops detected as commutative by DCA. As seen in the table, DCA uncovers potential parallelism that closely matches each of the dynamic techniques. The detection effectiveness of DCA is attributed to its ability to capture the effects of code that spans large regions with complex control flow, function calls and non-linear array accesses.

2) Detection of PLDS-Based Loops: While DCA is effective at detecting commutative array-based loops, its true potential as a unified analysis for parallelism discovery is best seen on PLDS-based loops. DCA is able to detect *automatically* as commutative a broad collection of popular PLDS-based loops from earlier compiler studies, whereas the baseline tools, both dynamic and static, hit their limits and *fail to detect any of them as parallel*. The focus of these earlier studies has been on profitably exploiting these loops on modern parallel architectures rather than detecting them as parallel. Unlike DCA, these studies rely on *ad hoc* manual methods to

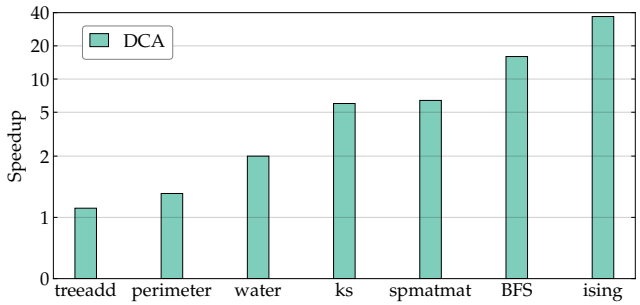

Figure 5. Overall speedup over sequential code achieved by DCA parallelization for PLDS loops. The parallel code generation techniques from Table II fail to detect parallelizable loops automatically.

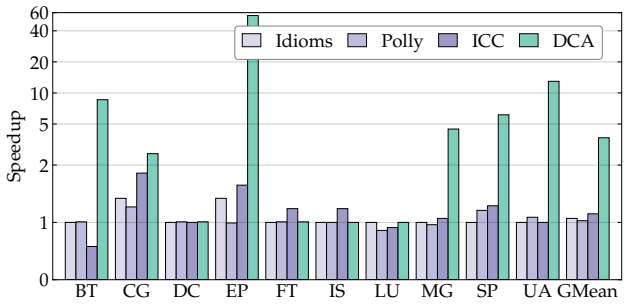

Figure 6. Overall speedup over sequential code achieved by IDIOMS, POLLY, ICC and DCA parallelization for NPB.

identify and utilize PLDS-based loops. The PLDS-based loops detected as commutative are summarized in Table II. The table also reports, for each loop, its origin, coverage and potential speedup found by *ad hoc* manual methods as described in the literature [37], [38], [40], [43], [44], [45], [46]. Remarkably, DCA detects commutative loops even in complex programs, such as *300.twolf* which contains doubly-nested linked list traversals, similar in nature to Fig. 1(b). Other loops, such as *treeadd*, employ a worklist traversal idiom akin to BFS presented in Fig. 2.

An interesting case is *429.mcf*, the only loop in Table II known *not* to be statically commutative. *429.mcf* performs a complicated tree traversal, accessing sibling and predecessor nodes, and contains a cross-iteration dependence. The dependence is not exercised by the test or the reference workloads, hence DCA reports it as commutative. Speculative parallelization approaches in the literature rely on the assumption that this dependence is infrequent to parallelize the loop profitably.

In summary, DCA's uniform approach is able to discover potential parallelism in loops such as PLDS traversals that are well beyond the limits of dynamic dependence-based analysis approaches. This opens a potential avenue for leveraging techniques aiming at efficient parallelism exploitation of complex loops, such as speculative parallelization.

3) Parallelization of PLDS-Based Loops: Our results demonstrate that DCA's simple parallelization scheme can yield speedups for PLDS-based loops (up to 36×), although it is not as widely effective as for array-based loops. Fig. 5 reports the parallelization speedup achieved by DCA for a selection of the PLDS-based loops found as commutative in Table II. The selection combines simple kernels (such as *spmatmat*, a sparse matrix-matrix multiplication) and loops from larger, more complex programs (such as *water-spatial*, an n-body wave simulation).

For the remaining loops in Table II, DCA's simple parallelization scheme does not yield significant speedups and requires more specialized parallel code generation techniques (last column of that table) for profitable exploitation. However, in the cases where DCA's simple parallelization is not effective, its analysis results might still be exploited by more sophisticated parallel code generation techniques such as those reported in Table II. Exploring the interaction between DCA's parallelism discovery and these parallelization techniques is part of future work.

C. Performance against Static Techniques

1) Detection of Array-Based Loops: Similarly to Table I, Table III reports the number of loops that each static baseline approach reports as parallelizable. It also presents a comparison of the *combined* results of all three static approaches (*Combined Static*) and the number of loops detected as commutative by DCA. DCA uncovers nearly twice as much potential parallelism (86% of all loops) as the combined static baseline (49%).

For five out of the ten benchmarks in the suite, DCA finds over 80% of the loops as potentially parallelizable, substantially more than the combined static baseline, which achieves less than 50% for the same benchmarks. ICC is more robust in detecting parallelizable loops than the other two baseline approaches. In some cases this is due to more aggressive inlining of pure (i.e., no side effects) functions. On the other hand, ICC is unable to detect complex reduction and histogram operations discovered by IDIOMS. DCA correctly identifies all the above loops which are executed as commutative.

In the rest of the NPB suite, DCA also achieves high detection scores, outperforming the combined baseline. *CG* contains a higher number of loops exhibiting cross-iteration dependences, which neither DCA nor the rest of the approaches detect. *MG* displays a somewhat unusual coding style with respect to the rest of suite, using I/O in several nested loops and contains a number of loops that the input workloads do not exercise. We have asserted that DCA can detect these loops as potentially parallelizable given the appropriate input conditions, but excluded them from our final results for consistency. Unsurprisingly, DCA detects the least number of potentially parallelizable loops for *DC*, which performs numerous I/O operations.

2) Parallelization of Array-Based Loops: Our results show that the simple parallelization scheme proposed in Section IV-C is generally effective for array-based loops. Overall, by parallelizing the profitable loops among those discovered

TABLE III
NPB LOOPS REPORTED AS PARALLELIZABLE BY THE BASELINE STATIC APPROACHES AND AS COMMUTATIVE BY DCA. "COMBINED STATIC" INDICATES THE RESULTS OF THE STATIC TECHNIQUES (IDIOMS [51], POLLY [52] AND ICC [53]) COMBINED.

Benchmark	Loops	IDIOMS		POLLY		ICC		Combined Static		DCA (this work)	
	(#)	(#)	(%)	(#)	(%)	(#)	(%)	(#)	(%)	(#)	(%)
BT	182	5	3	34	19	50	27	80	44	168	92
CG	47	9	19	8	17	23	49	25	53	33	70
DC	105	14	13	11	10	23	22	39	37	41	39
EP	9	2	22	2	22	3	33	4	44	6	67
FT	42	1	2	6	14	1	2	8	19	36	86
IS	16	7	44	3	19	3	19	11	69	12	75
LU	186	3	2	19	10	81	44	90	48	160	86
MG	81	8	10	5	6	21	26	32	40	48	59
SP	250	2	1	38	15	93	37	113	45	233	93
UA	479	23	5	43	9	180	38	209	44	466	97
Total	**1397**	**74**	**5**	**169**	**12**	**478**	**34**	**611**	**44**	**1203**	**86**

as commutative in Section V-C1, DCA achieves an average speedup of 3.6× (and up to 55.2×) over the sequential version of each NPB benchmark. Fig. 6 presents the speedup results for each benchmark.

Since profitability analysis is out of DCA's current scope, only the commutative loops deemed as profitable in the expert NPB implementation (and the hottest ones for the case where this information is not available) are selected for parallelization. Remarkably, DCA detects as commutative all data-parallel loops deemed profitable in the expert parallelization.

EP is a small kernel with a hot two-level loop nest performing an integral evaluation via pseudo-random trials. Parallelizing the outer loop which contains the complex reduction loop yields a speedup of 55.2×. DCA also achieves significant speedups for *BT*, *CG*, *MG*, *SP* and *UA* (8.6×, 2.6×, 4.5×, 6.1× and 13× respectively). This is attributed to DCA's ability to detect and exploit loops that extend across many lines of code, containing function calls and complex control flow.

For *DC*, *FT*, *IS* and *LU*, the relatively high number of detected commutative loops does not translate into profitable parallelism. For example, *DC* is an I/O intensive benchmark manipulating data at volumes much larger than a modern system's memory capacity, while *LU* contains dependences across hot function calls. Thus, these programs require higher level of synchronization or extended code restructuring for their efficient exploitation.

DCA (together with expert profitability analysis) consistently outperforms the baseline parallelization by IDIOMS, POLLY, and ICC, reported in Fig. 6. The same profitability analysis is applied for IDIOMS as for DCA, whereas for ICC and POLLY their optimal profitability analysis is used (Section V-A). ICC and IDIOMS are able to extract some latent loop parallelism in *CG* and *EP*, but are still outperformed by DCA. IDIOMS is able to exploit *EP*, but the effectiveness of parallelization is limited to the inner hot loop. The difference between ICC and DCA for *FT* and *IS* is due to the unavoidable compiler optimization differences: manually implementing the parallelization suggestions from ICC's reports for *IS* and compiling it in our framework results in the same performance that ICC obtains.

TABLE IV
DCA COVERS A SIGNIFICANT FRACTION OF EXECUTION TIME, PROVIDING HIGH PRECISION DETECTION RESULTS.

Bmk	Loops	DCA (this work)				Combined Static
		Found	False Positive	False Negative	Sequential Coverage	Sequential Coverage
	(#)	(#)	(#)	(#)	(%)	(%)
BT	182	168	0	0	100	36
CG	47	33	0	0	91	7
DC	105	41	0	0	0	0
EP	9	6	0	0	100	37
FT	41	36	0	0	91	42
IS	16	12	0	0	60	56
LU	186	160	0	0	84	56
MG	81	48	0	0	87	56
SP	250	233	0	0	94	77
UA	479	466	0	0	86	57

D. Aspects of Detection Profitability and Precision

In order to assess the accuracy of DCA's predictions, all loops were further analyzed semi-manually, employing expert algorithmic knowledge and a combination of targeted profiling and testing of dependences and computations.

The results (Table IV) show that all loops determined as commutative by DCA are indeed parallelizable (i.e., no false positives), following the spirit of the discussion in Section III. This result strengthens the significance of Tables I and III, as it confirms that DCA can indeed uncover many valid opportunities for parallelization on top of the baseline approaches.

Future work could improve DCA's effectiveness by applying combined tests for multiple inputs and exploring inputs leading to execution paths that might affect commutativity.

The loops found by DCA are significant in that they cover a considerable fraction of the total execution time for most benchmarks in NPB, with above 80% for eight out of the ten benchmarks in the suite and almost 100% for *BT* and *EP*. As shown by the two rightmost columns of Table IV, DCA consistently outperforms the combined static approaches. This agrees with the high detection rate shown in Table III and the parallelism that is profitably exploited as reported in Fig. 6.

Overall, DCA accurately detects a large amount of potentially relevant and profitable parallelism, beyond the capabilities of static analysis approaches.

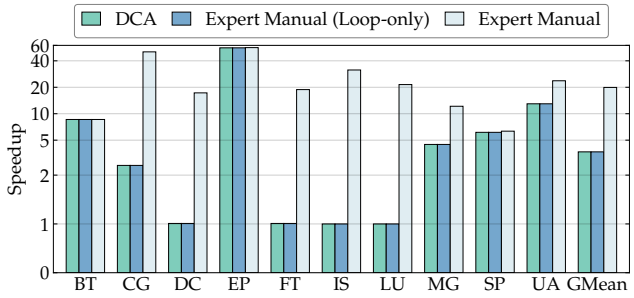

Figure 7. Overall speedup over sequential code achieved by DCA and Expert Manual (loop-only and whole program) parallelization for NPB. DCA matches the performance of manual parallelization by experts at the loop level.

E. Scope of Parallelization Beyond Loops

The evaluation has so far focused on the detection and parallelization of loops, where DCA succeeds in uncovering more hidden loop parallelism than several state-of-the-art approaches. There exists, however, parallelism that is not strictly confined within loops.

Fig. 7 compares the speedups of DCA on the NPB suite with those obtained by the data-parallel loops as parallelized by experts (*Expert Manual (Loop-only)*) and a full expert parallelization beyond single-loop data parallelism (*Expert Manual*). The results show that DCA succeeds in identifying and exploiting data-parallel loops and that opportunity remains for further parallelization beyond their scope.

DCA is successful in extracting all the available parallelism from *BT*, *EP* and *SP*. It is also fairly effective on *MG* and *UA*, where the performance discrepancy with the full expert parallelization is due to the fact that the latter exploits whole parallel sections, spanning across loops, thus improving locality and minimizing synchronization costs.

The rest of the benchmarks contain parallelism that is outside the loop-level focus of DCA. The full expert parallelization of these benchmarks uses additional algorithmic knowledge. *DC* and *FT* are largely restructured to take advantage of independent work-sharing. *LU* uses a pipeline pattern to overcome the bottleneck in its hottest loop nests. *CG* falls under the same category, however, DCA is able to exploit profitably some of its remaining loop-level parallelism.

The gap between loop-level parallelism and what can be achieved by taking a wider, structural view of program parallelism motivates further research on this topic. We believe that commutativity analysis can also play a key role in discovering and exploiting structured parallelism.

VI. Related Work

Code analysis supporting parallelization has a long history [54], [55], [56]. Existing commutativity analyses for parallelization have already been discussed in Section II.

A. Dynamic and Profile-Driven Analyses

Tracing and dynamic profiling have been used to capture application properties for subsequent exploitation in benchmark characterization [18], collaborative runtime verification [19] and dynamic program optimization [20]. Unsafe dependence profiling has been used to overcome the limits of static dependence analysis [8], [9], [6]. This is because statically proving the absence of dependences is generally undecidable [57], similarly to other facts in static analyses [58], [59], [23].

Such approaches, including [5], [6], [7], [8], [10], [11], [12], [13], [14], [15], [16], [22], combine dependence profiling information with static dependence analyses to gain additional information on *may*-dependences. SAMBAMBA [17] combines several techniques in a dynamic framework for automatic parallelization.

B. Parallelization in the Presence of Dependences

ALTER [60] is a dynamic technique which exploits *breakable dependences* for parallelization, extracting loop parallelism by reordering iterations or allowing stale reads. ALTER relies on the user and, partially, on a set of tests to infer potential annotations, and it has been evaluated on hand-picked loops, some of which have been manually parallelized. While we share this notion of iteration reordering and breakable dependences with ALTER, DCA derives its tests automatically using live-outs, it has been evaluated on standard benchmarks and demonstrates parallelization using existing techniques.

DSWP [61], [40] and HELIX [62] are both designed to derive parallel execution schedules for loops with data dependences. Some variants rely on additional hardware support for the fast communication of values and synchronization between threads, typically outperforming their software-only counterparts. While DSWP and HELIX focus on the exploitation of parallelism, DCA complements them by identifying sources of profitable parallelism.

[31] presents a technique to verify commutativity conditions, which are logical formulae that characterize when operations on a linked data structure commute. Code annotations for commutative functions are also proposed in parallelization frameworks by [63], as well as in GALOIS [64] and PARALAX [65]. [66] presents a commutativity-based programming model called COMMSET (Commutative Set) and its associated compiler technology. FRACTAL SYMBOLIC ANALYSIS uses a commute operation to verify correctness of restructuring transforms, which have a user-provided, rule-based description in its implementation [67].

C. Speculative Parallelization

Speculative parallelization techniques optimistically execute potentially independent regions of code. DCA avoids the usual costs associated with speculation during runtime, and even though user interaction might be required during analysis, in practice, our technique offers high precision.

The LRPD test [21] is one of the earliest and most influential pieces of work on speculative parallelization. Since then, research has moved on to explore hardware support, compiler extensions and software behavior characterization [68]. For PLDS-based programs, [69] extends an earlier approach to software-based parallelization that separates the program states

to speculative and non-speculative by adding a mapping table to keep track of accesses between these sections. [70] analyzes program behavior to detect potential sequential sections that may be executed in parallel.

We consider speculative techniques orthogonal to DCA's detection capabilities, that can be used on top of it for further exploitation.

D. Parallelization with User Interaction

Parallelizing systems relying on user-provided feedback may require that input at various stages [71]; from annotating code [46] to validating the proposed transformations [72], [8]. In all cases the user is responsible for ensuring correctness. Our system adopts the latter approach, similar to [8], but extending the potential candidates to PLDS-based loops.

E. Analyzing Pointer-Linked Data Structures

[73] presents an approach that is based on programming language annotations to exploit parallel execution of PLDSs. Later pioneering work by Laurie Hendren has focused on the automatic analysis and parallelization of such programs [47]. Efforts moved from pointer analysis to a programming model where programmers describe their intentions [74]. The next wave of research has focused on shape analysis [75], [76], which seeks to discover and verify (shape) properties of PLDSs statically. More recent work [77] has investigated ways of optimizing the data layout of PLDSs.

VII. Conclusion

We have developed DCA, a novel analysis for identifying parallelizable loops in sequential legacy code, which relies on liveness-based commutativity.

We extensively evaluated DCA against five state-of-the-art parallelization approaches on array-based benchmarks, where it identifies as many parallelizable loops as two dynamic techniques, and nearly twice as many as three static tools *combined*. DCA also uncovers parallelism in irregular loops dominated by PLDS structures, in the same *uniform* manner as for array-based loops, *where all other analyses tested fall short*. Experiments show that DCA results in an average speedup of $3.6\times$ across NPB (and up to $55\times$) on a 72-core host, and up to $36.9\times$ for the PLDS-based loops.

Future work will extend the scope of DCA to code regions of any granularity with the ultimate goal to support the detection of parallel algorithmic skeletons in legacy code.

References

[1] K. Pingali, "Why compilers have failed and what can we do about it," Houston, Texas, USA, October 7 - 9, 2010.

[2] S. Amarasinghe, "Why compilers have failed to support HPC programmers and what can we do about it," in *ASCR Programming Challenges Workshop*, 2011.

[3] S. Apostolakis, Z. Xu, G. Chan, S. Campanoni, and D. I. August, "Perspective: A sensible approach to speculative automatic parallelization," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. Lausanne, Switzerland: Association for Computing Machinery, Mar. 2020, pp. 351–367.

[4] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*, 1st ed. San Francisco: Morgan Kaufmann Publishers In, Mar. 2000.

[5] T. Austin and G. Sohi, "Dynamic dependency analysis of ordinary programs," in *[1992] Proceedings the 19th Annual International Symposium on Computer Architecture*, May 1992, pp. 342–351.

[6] K.-F. Faxén, K. Popov, S. Jansson, and L. Albertsson, "Embla - data dependence profiling for parallel programming," in *2008 International Conference on Complex, Intelligent and Software Intensive Systems*, Mar. 2008, pp. 780–785.

[7] X. Zhang, A. Navabi, and S. Jagannathan, "Alchemist: A transparent dependence distance profiling infrastructure," in *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '09. USA: IEEE Computer Society, Mar. 2009, pp. 47–58.

[8] G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle, "Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine-learning based mapping," *SIGPLAN Not.*, vol. 44, no. 6, pp. 177–187, Jun. 2009.

[9] Z. Li, R. Atre, Z. Huda, A. Jannesari, and F. Wolf, "Unveiling parallelization opportunities in sequential programs," *J. Syst. Softw.*, vol. 117, no. C, pp. 282–295, Jul. 2016.

[10] M. Kim, H. Kim, and C.-K. Luk, "SD3: A Scalable Approach to Dynamic Data-Dependence Profiling," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '43. USA: IEEE Computer Society, Dec. 2010, pp. 535–546.

[11] ——, "Prospector: A dynamic data-dependence profiler to help parallel programming," in *Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism*. Berkeley, CA: USENIX Association, Jun. 2010.

[12] D. Jeon, S. Garcia, C. Louie, S. Kota Venkata, and M. B. Taylor, "Kremlin: Like gprof, but for parallelization," *SIGPLAN Not.*, vol. 46, no. 8, pp. 293–294, Feb. 2011.

[13] R. Vanka and J. Tuck, "Efficient and accurate data dependence profiling using software signatures," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12. San Jose, California: Association for Computing Machinery, Mar. 2012, pp. 186–195.

[14] Y. Sato, Y. Inoguchi, and T. Nakamura, "Whole program data dependence profiling to unveil parallel regions in the dynamic execution," in *2012 IEEE International Symposium on Workload Characterization (IISWC)*, Nov. 2012, pp. 69–80.

[15] H. Yu and Z. Li, "Fast loop-level data dependence profiling," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12. San Servolo Island, Venice, Italy: Association for Computing Machinery, Jun. 2012, pp. 37–46.

[16] ——, "Multi-slicing: A compiler-supported parallel approach to data dependence profiling," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. Minneapolis, MN, USA: Association for Computing Machinery, Jul. 2012, pp. 23–33.

[17] K. Streit, C. Hammacher, A. Zeller, and S. Hack, "Sambamba: A Runtime System for Online Adaptive Parallelization," in *Compiler Construction*, ser. Lecture Notes in Computer Science, M. O'Boyle, Ed. Berlin, Heidelberg: Springer, 2012, pp. 240–243.

[18] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge, "Dynamic metrics for java," in *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, ser. OOPSLA '03. Anaheim, California, USA: Association for Computing Machinery, Oct. 2003, pp. 149–168.

[19] E. Bodden, L. Hendren, P. Lam, O. Lhoták, and N. A. Naeem, "Collaborative runtime verification with tracematches," in *Proceedings of the 7th International Conference on Runtime Verification*, ser. RV'07. Vancouver, Canada: Springer-Verlag, Mar. 2007, pp. 22–37.

[20] M. Berndl and L. Hendren, "Dynamic profiling and trace cache generation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, ser. CGO '03. San Francisco, California, USA: IEEE Computer Society, Mar. 2003, pp. 276–285.

[21] L. Rauchwerger and D. Padua, "The LRPD Test: Speculative Run-time Parallelization of Loops with Privatization and Reduction Parallelization," in *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, ser. PLDI '95. New York, NY, USA: ACM, 1995, pp. 218–232.

[22] P. Wu, A. Kejariwal, and C. Caşcaval, "Compiler-Driven Dependence Profiling to Guide Program Parallelization," in *Languages and Compilers*

for Parallel Computing, J. N. Amaral, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, vol. 5335, pp. 232–248.

[23] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, "In Defense of Soundiness: A Manifesto," *Commun ACM*, vol. 58, no. 2, pp. 44–46, Jan. 2015.

[24] T. J. Edler von Koch and B. Franke, "Variability of data dependences and control flow," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2014, pp. 180–189.

[25] N. Murphy, T. Jones, R. Mullins, and S. Campanoni, "Performance implications of transient loop-carried data dependences in automatically parallelized loops," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC 2016. Barcelona, Spain: Association for Computing Machinery, Mar. 2016, pp. 23–33.

[26] M. C. Rinard and P. C. Diniz, "Commutativity analysis: A new analysis technique for parallelizing compilers," *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 6, pp. 942–991, Nov. 1997.

[27] F. Aleen and N. Clark, "Commutativity analysis for software parallelization: Letting program transformations see the big picture," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV. New York, NY, USA: ACM, 2009, pp. 241–252.

[28] T. J. K. E. von Koch, S. Manilov, C. Vasiladiotis, M. Cole, and B. Franke, "Towards a compiler analysis for parallel algorithmic skeletons," in *Proceedings of the 27th International Conference on Compiler Construction*, ser. CC 2018. New York, NY, USA: ACM, 2018, pp. 174–184.

[29] K. D. Cooper and L. Torczon, *Engineering a Compiler*, 2nd ed. Amsterdam ; Boston: Elsevier/Morgan Kaufmann, 2012.

[30] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali, "Lonestar: A suite of parallel irregular programs," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, Apr. 2009, pp. 65–76.

[31] D. Kim and M. C. Rinard, "Verification of semantic commutativity conditions and inverse operations on linked data structures," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 528–541.

[32] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv. CSUR*, vol. 51, no. 3, p. 50, Jul. 2018.

[33] S. Manilov, C. Vasiladiotis, and B. Franke, "Generalized profile-guided iterator recognition," in *Proceedings of the 27th International Conference on Compiler Construction - CC 2018*. Vienna, Austria: ACM Press, 2018, pp. 185–195.

[34] H. L. A. van der Spek, E. M. Bakker, and H. A. G. Wijshoff, "SPARK00: A Benchmark Package for the Compiler Evaluation of Irregular/Sparse Codes," *ArXiv08053897 Cs*, May 2008.

[35] B. Pottenger and R. Eigenmann, "Idiom recognition in the Polaris parallelizing compiler," in *Proceedings of the 9th International Conference on Supercomputing*, ser. ICS '95. Barcelona, Spain: Association for Computing Machinery, Jul. 1995, pp. 444–448.

[36] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput Arch. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006.

[37] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August, "Decoupled Software Pipelining with the Synchronization Array," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 177–188.

[38] R. Rangan, "Pipelined multithreading transformations and support mechanisms," PhD Thesis, Princeton University, USA, 2007.

[39] J. L. Henning, "Spec cpu2000: Measuring cpu performance in the new millennium," *Computer*, vol. 33, no. 7, pp. 28–35, Jul. 2000.

[40] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August, "Parallel-stage decoupled software pipelining," in *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '08. Boston, MA, USA: Association for Computing Machinery, Apr. 2008, pp. 114–123.

[41] T. M. Austin, S. E. Breach, and G. S. Sohi, "Efficient detection of all pointer and array access errors," *SIGPLAN Not.*, vol. 29, no. 6, pp. 290–301, Jun. 1994.

[42] M. C. Carlisle, A. Rogers, J. H. Reppy, and L. J. Hendren, "Early experiences with Olden," in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Aug. 1993, pp. 1–20.

[43] M. Feng, C. Lin, and R. Gupta, "PLDS: Partitioning linked data structures for parallelism," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 38:1–38:21, Jan. 2012.

[44] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The tao of parallelism in algorithms," *SIGPLAN Not.*, vol. 46, no. 6, pp. 12–25, Jun. 2011.

[45] P. Kraft, A. Waterland, D. Y. Fu, A. Gollamudi, S. Szulanski, and M. Seltzer, "Automatic parallelization of sequential programs," *ArXiv180907684 Cs*, Jul. 2018.

[46] J. M. M. Caamaño, A. Sukumaran-Rajam, A. Baloian, M. Selva, and P. Clauss, "APOLLO: Automatic speculative POLyhedral Loop Optimizer," in *IMPACT 2017 - 7th International Workshop on Polyhedral Compilation Techniques*, Jan. 2017, p. 8.

[47] R. Ghiya and L. J. Hendren, "Putting pointer analysis to work," in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '98. New York, NY, USA: ACM, 1998, pp. 121–133.

[48] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2016, pp. 101–111.

[49] D. H. Bailey, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, S. K. Weeratunga, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, and T. A. Lasinski, "The NAS parallel benchmarks," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing - Supercomputing '91*. Albuquerque, New Mexico, United States: ACM Press, 1991, pp. 158–165.

[50] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–.

[51] P. Ginsbach and M. F. P. O'Boyle, "Discovery and exploitation of general reductions: A constraint based approach," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, ser. CGO '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 269–280.

[52] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly — Performing Polyhedral Optimizations on a Low-Level Intermediate Representation," *Parallel Process. Lett.*, vol. 22, no. 04, p. 1250010, Dec. 2012.

[53] "Intel® C++ Compiler," https://software.intel.com/en-us/c-compilers, Aug. 2020.

[54] L. Lamport, "The Parallel Execution of DO Loops," *Commun ACM*, vol. 17, no. 2, pp. 83–93, Feb. 1974.

[55] S. P. Midkiff, "Automatic Parallelization: An Overview of Fundamental Compiler Techniques," *Synthesis Lectures on Computer Architecture*, vol. 7, no. 1, pp. 1–169, Jan. 2012.

[56] S. Prema, R. Nasre, R. Jehadeesan, and B. Panigrahi, "A study on popular auto-parallelization frameworks: A study on popular auto-parallelization frameworks," *Concurrency Computat Pract Exper*, vol. 31, no. 17, p. e5168, Sep. 2019.

[57] T. Reps, "Undecidability of context-sensitive data-dependence analysis," *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 1, pp. 162–186, Jan. 2000.

[58] W. Landi, "Undecidability of static analysis," *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 4, pp. 323–337, Dec. 1992.

[59] A. Charlesworth, "The Undecidability of Associativity and Commutativity Analysis," *ACM Trans Program Lang Syst*, vol. 24, no. 5, pp. 554–565, Sep. 2002.

[60] A. Udupa, K. Rajan, and W. Thies, "ALTER: Exploiting Breakable Dependences for Parallelization," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 480–491.

[61] G. Ottoni, R. Rangan, A. Stoler, and D. I. August, "Automatic Thread Extraction with Decoupled Software Pipelining," in *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 38. Washington, DC, USA: IEEE Computer Society, 2005, pp. 105–118.

[62] S. Campanoni, T. Jones, G. Holloway, V. J. Reddi, G.-Y. Wei, and D. Brooks, "HELIX: Automatic parallelization of irregular programs for chip multiprocessing," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization - CGO '12*. San Jose, California: ACM Press, 2012, p. 84.

[63] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August, "Revisiting the sequential programming model for multi-core," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40. USA: IEEE Computer Society, Dec. 2007, pp. 69–84.

[64] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 211–222.

[65] H. Vandierendonck, S. Rul, and K. De Bosschere, "The Paralax Infrastructure: Automatic Parallelization with a Helping Hand," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 389–400.

[66] P. Prabhu, S. Ghosh, Y. Zhang, N. P. Johnson, and D. I. August, "Commutative set: A language extension for implicit parallel programming," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 1–11.

[67] V. Menon, K. Pingali, and N. Mateev, "Fractal symbolic analysis," *ACM Trans. Program. Lang. Syst.*, vol. 25, no. 6, pp. 776–813, Nov. 2003.

[68] A. Estebanez, D. R. Llanos, and A. Gonzalez-Escribano, "A Survey on Thread-Level Speculation Techniques," *ACM Comput. Surv.*, vol. 49, no. 2, pp. 22:1–22:39, Jun. 2016.

[69] C. Tian, M. Feng, and R. Gupta, "Supporting speculative parallelization in the presence of dynamic data structures," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '10. New York, NY, USA: Association for Computing Machinery, Jun. 2010, pp. 62–73.

[70] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang, "Software behavior oriented parallelization," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. San Diego, California, USA: Association for Computing Machinery, Jun. 2007, pp. 223–234.

[71] S.-W. Liao, A. Diwan, R. P. Bosch, A. Ghuloum, and M. S. Lam, "SUIF Explorer: An interactive and interprocedural parallelizer," in *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '99. Atlanta, Georgia, USA: Association for Computing Machinery, May 1999, pp. 37–48.

[72] O. Zinenko, S. Huot, and C. Bastoul, "Visual Program Manipulation in the Polyhedral Model," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 1, pp. 16:1–16:25, Mar. 2018.

[73] R. Gupta, "SPMD execution of programs with pointer-based data structures on distributed-memory machines," *Journal of Parallel and Distributed Computing*, vol. 16, no. 2, pp. 92–107, Oct. 1992.

[74] J. Hummel, A. Nicolau, and L. J. Hendren, "A language for conveying the aliasing properties of dynamic, pointer-based data structures," in *Proceedings of the 8th International Symposium on Parallel Processing*. USA: IEEE Computer Society, Apr. 1994, pp. 208–216.

[75] N. Rinetzky and M. Sagiv, "Interprocedural Shape Analysis for Recursive Programs," in *Compiler Construction*, ser. Lecture Notes in Computer Science, R. Wilhelm, Ed. Berlin, Heidelberg: Springer, 2001, pp. 133–149.

[76] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang, "Shape Analysis for Composite Data Structures," in *Proceedings of the 19th International Conference on Computer Aided Verification*, ser. CAV'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 178–192.

[77] H. L. A. van der Spek, C. W. M. Holm, and H. A. G. Wijshoff, "A Compilation Framework for the Automatic Restructuring of Pointer-Linked Data Structures," in *High-Performance Scientific Computing: Algorithms and Applications*, M. W. Berry, K. A. Gallivan, E. Gallopoulos, A. Grama, B. Philippe, Y. Saad, and F. Saied, Eds. London: Springer, 2012, pp. 97–122.